Chapter 4. TECHNIQUES TO AID MOTION SPECIFICATION

Any technique short of requiring the animator to specify each and every pixel value at each and every frame attempts, to some extent, to abstract out the important features of the animation. Of course, selecting the important feature of an animation is a very subjective thing to do. Consequently, the appropriate feature for one kind of animation will not be the appropriate feature for another kind of animation. Thus, this section refers to the various techniques in which the animator is responsible for specifying most of the information. That is, the animator is working at a fairly low level of abstraction. These techniques differ from model-based techniques in that there is a more direct relationship between the information provided by the animator and the resulting motion. In contrast, in model-based animation the animator works at a high level of abstraction and, as a result, gives up more control of the specific motion, but less information is required from the animator.

4.1 Interpolation

At the foundation of almost all animation is the interpolation of values. The simplest case is interpolating the position of a point in space. Even this is non-trivial to do correctly and requires some discussion of several issues: the appropriate parameterization of position, the appropriate interpolating function, and maintaining the desired control of the interpolation over time.

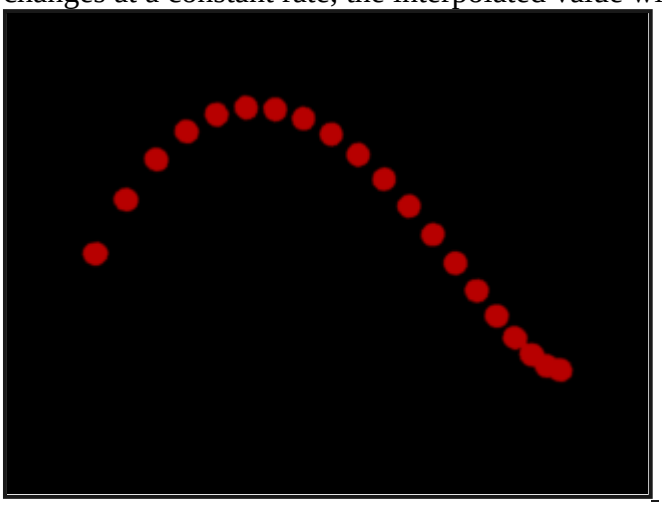
Often, an animator has a list of values associated with a given parameter at specific frames (called *key frames* or *keys*) of the animation. The question to be answered is how best to generate the values of the parameter for the frames between the key frames. The parameter to be interpolated may be a coordinate of the position of an object, or a joint angle of an appendage of a robot, or the transparency attribute of an object, or any other parameter used in the display and manipulation of computer graphic elements.

Appendix A contains a detailed discussion of specific interpolation techniques. Here, we will discuss the issues that determine how to choose the most appropriate interpolation technique and apply it in the production of an animated sequence. One of the first decisions to make is whether the given values represent actual values that the parameter should have at the key frames (*interpolation*), or whether they are meant merely to control the interpolating function and do not represent actual values the parameter will

assume (*approximation*). Other issues that influence which interpolation technique to use include how smooth the resulting function needs to be (i.e. *continuity*), how much computation you can afford to do (*order of interpolating polynomial*, and whether *local or global control* of the interpolating function is required. See Appendix A for a discussion of all of these issues. Suffice to say at this point, the *Catmull-Rom spline* is often used in animation path movement because it is an interpolating spline and requires no additional information from the user besides the points that the path is to pass through. *Parabolic blending* is an often-overlooked technique that also affords local control and is interpolating. See Mortenson's book or the book by Rogers and Adams for discussion of such curves.

We will assume that an interpolating technique has been chosen and that a function P(t) has been selected which, for a given value of t, will produce a value, i.e., p = P(t). If position is being interpolated then three functions are used in the following manner. The x, y and z coordinates for the positions at the key frames are specified. Key frames are associated with specific values of the time (t) parameter. The x, y and z coordinates are considered independently. For example, the points (x,t) are used as control points for the interpolating curve so that X=Px(t), where P denotes an interpolating function and the subscript x is used to denote that this specific curve was formed using the x coordinates of the key positions. Similarly, Y=Py(t) and Z=Pz(t) are formed so that at any time, t, a position (x,y,z) can be produced.

Varying the parameter of interpolation, in this case t, by a constant amount, does not mean that the resulting values, in this case Euclidean position, will vary by a constant amount. This means that just because t changes at a constant rate, the interpolated value will not necessarily, in fact seldom, have a constant speed.



In order to ensure a constant speed for the interpolated value, the interpolating function has to be parameterized by arc length, i.e., distance along the curve of interpolation. Some type of reparameterization by arc length should be performed for most applications. Usually this reparameterization can be approximated without adding undue overhead or complexity to the interpolating function.

Reparameterization by Arc Length

Assume that we have a spline defined by P(t). As noted above, we have no guarantee that the variance in t is directly related to the distance traveled along the curve. That is to say, that the distance traveled from P(0) to P(.2) is not necessarily twice as far as the distance traveled along the curve from P(.2) to P(.3). In order to establish this relationship we have to reparameterize the interpolating spline by arc length or some scalar of arc length. There are various ways to approach this. Analytically, arc length is defined as:

$$L = \int_{u1 \text{ to } u2} \sqrt{pu \cdot pu}\, du$$

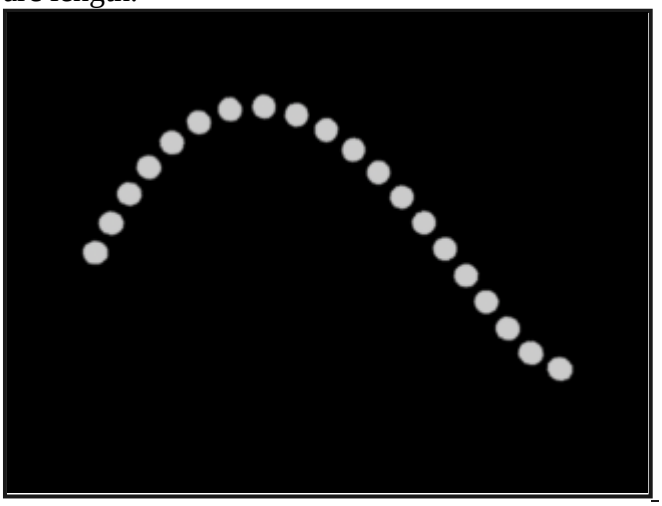
We use Gaussian quadrature to reduce the integral to

$$L = \text{summation}_{1 \text{ to } n} \text{ of } w_i * f(u_i)$$

where n is the number of sample points, w_i are the weight values, and u_i are the sampled values. See Mortenson's book, pp. 299-300. The u's can be normalized to the range zero to one and tables of weights and sample values can be found in tables. See *Numerical Recipes* for a discussion of Gaussian Quadrature.

Most of the time, the curves that arise in computer animation applications are not analytically reparameterizable by arc length. They must be reparameterized numerically. A simple, but somewhat inaccurate, approach to this reparameterization is to precompute a table of values which relates the original parameter with an arc length parameter. The number of entries in the table depends on the accuracy with which the arc length must be computed. This is determined by the application. The function is evaluated at n equidistant parameter values (equidistant in parameter space, e.g., t = 0.0, 0.01, 0.02, 0.03, etc.). N should be sufficiently large to ensure that the resulting arc lengths are within tolerance; this will become clear as the technique is described.

Using the parameter values and corresponding values of the function, a table is built which records accumulated linear distances between the computed points. Entries are then made in the table which record normalized distances along the curve. These entries are monotonic, increasing from zero to one, and represent the arc length parameterization (a scaled version of the arc length). This table can be precomputed before being used in the animation. This table can be used to determine the functional parameter needed to produce a point along the curve that corresponds to arc length and effectively parameterizes the function by arc length.



An alternative technique is computationally more efficient but programmatically more difficult. This technique uses adaptive Gaussian quadrature to determine the value of the function at the point along the curve that corresponds to the given arc length. See Brian Guenter's article.

4.2 Ease-In/Ease-Out

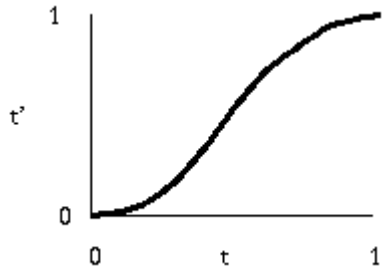
Note the difference between interpolating position along a curve and the speed along which the interpolation proceeds (consider the interpolating parameter to be 'time'). For example you can do linear interpolation in space but cubic interpolation of the distance with respect to the time parameter. If t is a parameter that goes between 0 and 1 and if the curve has been parameterized by arc length, then ease-in/ease-out can be implemented by a function t'=ease(t) so that as t varies uniformly between 0 and 1, t' will start at 0 slowly increasing in value and gaining speed until the middle values and then decelerating at

it approaches 1. t' is then used as the interpolation parameter in whatever function produces spatial values (again, assumed to already be parameterized by arc length).

Sine Interpolation

One easy way to implement ease-in/ease-out is to use the section of the sine curve from minus PI/2 to plus PI/2 as the ease(t) function:

t' = ease(t) = (sin(t*PI-PI/2)+1)/2



In this function, **t** is the input which is to be varied uniformly from zero to one (e.g., 0.0, 0.1, 0.2, 0.3, ...). The **t'** is the output which also goes from zero to one, but does so by starting off slow, speeding up, and then slowing down. For example, at t = .25, t' = .1465, and at t = .75, t' = .8535. With the sine/cosine ease-in/ease-out function presented here, the 'speed' of t' with respect to t is never constant for an interval but, rather, is always accelerating or decelerating.

Another method is to have the user specify times t1 and t2. A sinusoidal curve is used for velocity to implement an acceleration from time 0 to t1. A sinusoidal curve is also used for velocity to implement deceleration from time t2 to 1. Between times t1 and t2, a constant velocity is used. This is done by taking a parameter t in the range 0 to 1 and remapping it into that range according to the above velocity curves to get a new parameter rt. So as t varies uniformly from 0 to 1, rt will accelerate from 0, then maintain a constant parametric velocity and then decelerate back to 1.

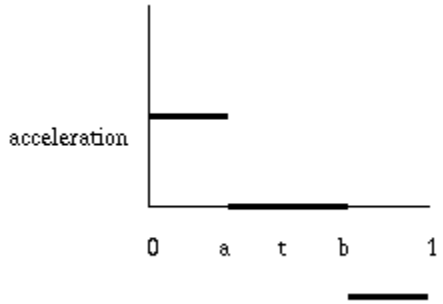
```
double ease(t,t1,t2)
double   t,t1,t2;
{
  double  rt,s,e1,e2,nt,rt;

  e1 = t1*2/PI;               /* distance after t1 acceleration */
  e2 = (1-t2)*2/PI;           /* distance after (1-t2) deceleration */

  if (t < t1) {                 /* if in acceleration stage */
    nt = t/t1;                /* fraction into acceleration */
    s = sin(-PI/2+nt*PI/2)+1; /* use sin quadrant for acceleration */
    rt = s*e1;                /* distance after s */
  }
  else if (t > t2) {
    nt = (t-t2)/(1.0-t2);     /* fraction into deceleration */
    s = sin(nt*PI/2);         /* use sine quadrant for acceleration */
    rt = e1 + (t2-t1) + s*e2; /* distance after s */
  }
  else {
    rt = e1 + t - t1;         /* distance after t */
  }

  rt = rt/(e1+(t2-t1)+e2);    /* scale back into 0:1 range */
```

```
    return(rt);
}
```

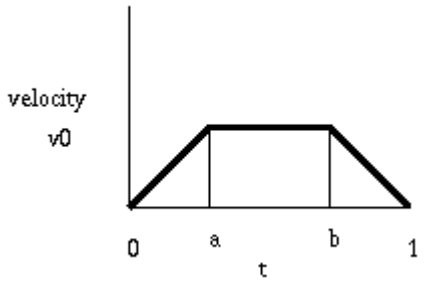
Constant Acceleration: Parabolic Ease-In/Ease-Out

An alternative approach and one that avoids the transcendental function evaluation, or corresponding table look-up and interpolation, is to establish basic assumptions about the acceleration and, from there, integrate to get the resulting interpolation function.

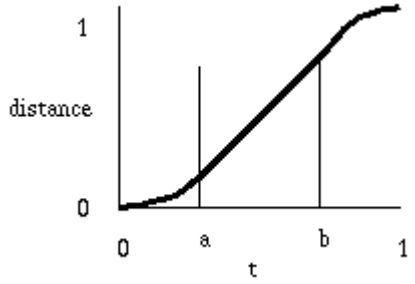
The default case of no ease-in/ease-out would produce a velocity curve that is a horizontal straight line of v0 as t goes from 0 to 1. The distance covered would be ease(1) = v0*1. To implement an ease-in/ease-out function, assume constant acceleration and deceleration at the beginning and end of the motion, and zero acceleration during the middle of the motion.



Integrate to get velocity. This produces a linear ramp for accelerating and decelerating velocities



and a distance function with parabolic sections at both ends.



(*Distance* in this case is in parametric space, or t-space, and is the distance covered by the output value of *t*.

For a given range of t = [0,1], the user specifies times to control acceleration and deceleration: a and b. Acceleration occurs from time 0 to time a. Deceleration occurs from time b to time 1.

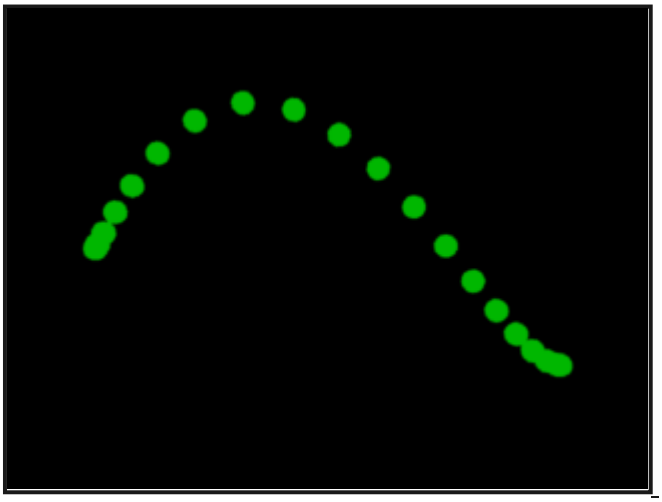
```
  t' = ease(t)
  v(t)  = v0*t/a                for t = [0,a]
        = v0                        t = [a,b]
        = v0 - (t-b)*v0/(1-b)       t = [b,1]
        = (1-t)*v0/(1-b)
```

where the distance covered must be equal to the distance covered in the default case. Of course the time can be renormalized into whatever range the user is interested in. For now, we will use a normalized distance of 1 to solve for v0 with the familiar *distance = average-speed*time*:

```
  1 = a*v0/2 + (b-a)*v0 + (1-b)*v0/2;
or
  v0 = 2/(b-a+1)
```

Now, integrating v(t) to get distance as a function of t:

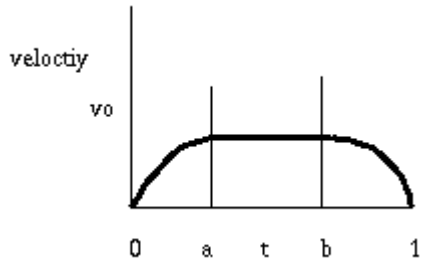
```
  d = v0*t**2/(2*a)                                    for t = [0,a]
    = v0*a/2+(t-a)*v0                                      t = [a,b]
    = v0*a/2+(b-a)*v0 + (t-t**2/2-b+b**2/2)*v0/(1-b)       t = [b,1]

  double ease(t,a,b)
  double  t,t1,t2;
  {
    double  v0,a1,a2;

    v0 = 2/(1+b-a);          /* constant velocity attained */
    if (t&lta) {
      d = v0*t*t/(2*a);
    }
    else {
      d = v0*a/2;
      if (t&ltb) {
        d += (t-a)*v0;
      }
      else {
        d += (b-a)*vo
        d += (t-t*t/2-b+b*b/2)*vo/(1-b);
      }
    }
    return(d);
  }
```



Sine Acceleration/Deceleration with Intermediate Constant Speed

Another method is to have the user specify times a and b, similar to the constant acceleration technique described above. However, in this case a segment of the sine curve is used to control the velocity acceleration and deceleration. Between times a and b, a constant velocity is used.

The development of this is similar to that for constant acceleration. A velocity curve is constructed from sin segments with interior constant velocity. The constant velocity, v0, is unknown.

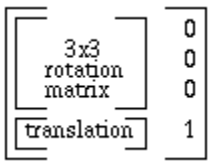


The distance covered can be computed as a function of v0. A normalized distance of one can be used and v0 can be solved for. From this the equations are integrated and equations for the distance covered can be derived (and is left as an exercise for the reader).

4.3 Orientation Representation and Interpolation

A common problem to address in computer animation is how to represent the position and orientation of an object in space, and how to change this representation over time to produce animation. A typical scenario is one in which the user specifies two such representations and the computer is used to interpolate intermediate positions thus producing animated motion. We will assume that there is no scaling involved, non-uniform or otherwise, so we are referring to rigid-body transformations.

The first obvious choice for representing the orientation and position of an object is by the 4x4 transformation matrix. A user can interactively rotate and translate an object to produce a compound 4x4 transformation matrix. In such a matrix the upper left 3x3 matrix represents a rotation to apply to the object and the first three elements of the fourth row represent the translation. (Assuming points are represented by row vectors which are post-multiplied by the transformation matrix.) No matter how the 4x4 transformation matrix was formed (no matter what order the transformations were given by the user, such as rotate, translate, rotate, rotate, translate, rotate), the final 4x4 transformation matrix produced by concatenating all of the individual transformation matrices in the specified order will result in a 4x4 matrix which specifies the final position of the object by a 3x3 rotation matrix followed by a translation.



The conclusion is that the rotation can be interpolated independent from the translation. (For now, we will

just consider linear interpolation although higher order interpolations are possible. See Appendix A.)

Now consider two such transformations that the user has specified as key positions that the computer should interpolate between. While it should be obvious that interpolating the translations is straightforward, it is not clear at all that the interpolating of the rotations will produce intuitive results. In fact, it is the objective of this introduction to show that interpolation of orientations can be a problem. A property of 3x3 rigid-body rotation matrices is that the rows and columns are orthonormal (unit length and perpendicular to each other). Simple linear interpolation between the nine pairs of numbers which make up the two 3x3 rotation matrices to be interpolated will not produce intermediate 3x3 matrices which are orthonormal and are, therefore, not rigid-body rotations. It should be pretty easy to see that interpolating a rotation of +90 degrees about the y-axis with a rotation of -90 degrees about the y-axis results in transformations which are nonsense. There are alternative representations which are more useful than transformation matrices in performing such interpolations.

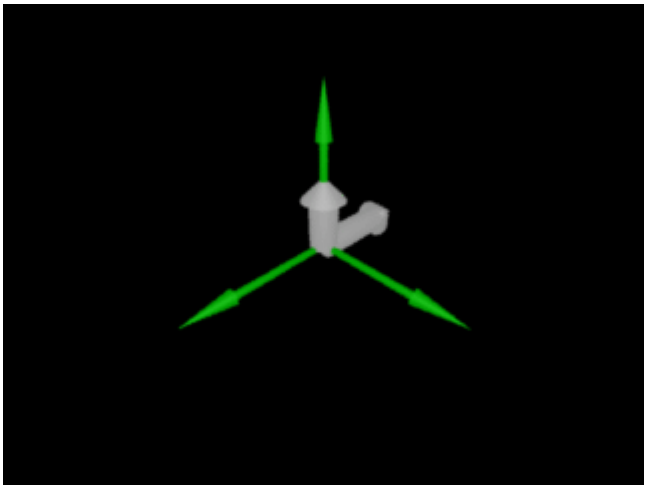
Another problem with using transformation matrices to represent orientations is that, if incremental rotations are accumulated, errors can arise in such a way that an invalid representation results. A fixed-angle representation avoids the invalid representation problem (although it, too, is subject to errors as shown below.)

Fixed-Angle Representation

Fixed-angle representation really means 'angles about fixed-axes'. A fixed order of three rotations is implied, such as x-y-z. This means that orientation is given by a set of three ordered angles, first around x, then around y, then around z. There are many possible orderings of the rotations and, in fact, it is not even necessary to use all three coordinate axes. For example, x-y-x is a feasible set of rotations. The only ones that don't make sense are those orderings in which an axis immediately follows itself such as x-x-y.

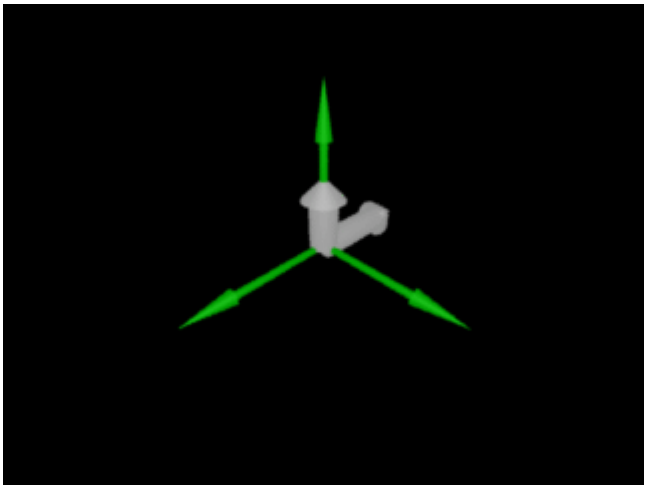
In any case, the main point is that the orientation of an object is given by three angles, such as (10,45,90). In this example, the orientation represented is the one obtained by rotating the object first about the x-axis by 10 degrees, then about the y-axis by 45 degrees, then about the z-axis by 90 degrees. We will use the following notation to represent such a sequence of rotations: Rx(10) Ry(45) Rz(90) (points are post-multiplied by transformation matrices).

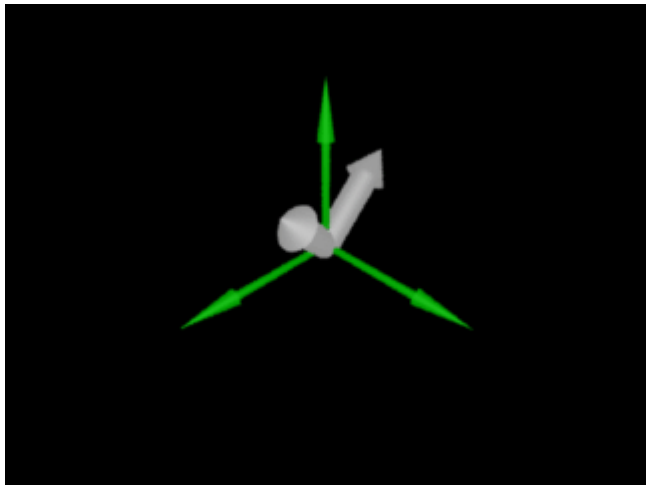
The problem with using this scheme is that two of the axes of rotation can essentially line up on top of each other when you consider what a slight change in value does from a given representation. For example, consider the orientation represented by (0,90,0) and how slight changes in the parameters will effect the object from this orientation.



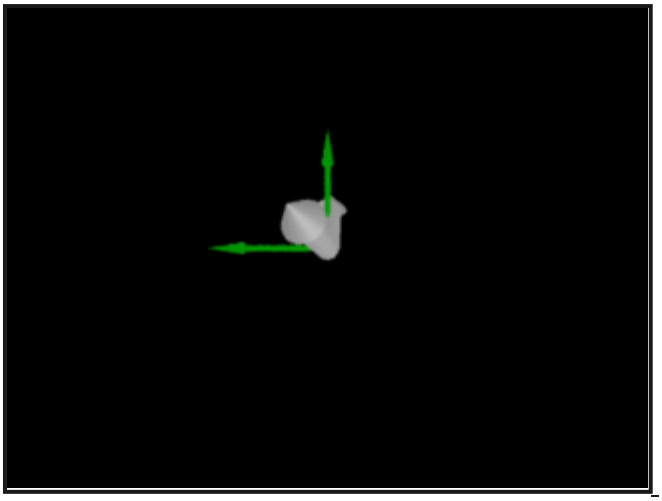
A slight change of the third parameter will rotate the object slightly about the z-axis. However, note that a slight change of the first parameter will also rotate the object slightly about the z-axis because the 90 degree y-axis rotation has essentially made the first axis of rotation align with the third axis of rotation. This is what is called **gimbal lock**. From the initial position (0,90,0) the object can no longer be rotated about the global x-axis by a simple change in its orientation representation (actually, the representation that will effect such an orientation is (90,90+epsilon,90) - not very intuitive!).

This same problem makes interpolation between key positions problematic in some cases. Consider the key orientations of (0,90,0) and (90,45,90).





The second orientation is a 45 degree x-axis rotation from the first position. However, as discussed above, the object can no longer directly rotate about the x-axis from the first key orientation because of the 90-degree y-axis rotation. Direct interpolation of the key orientation representations would produce (45,67.5,45) which is very different from the (90,22.5,90) orientation that is desired). Below is the half-way image produced when viewing from the negative z-axis toward the origin.



In its favor, the Fixed-Angle Representation is compact, intuitive and easy to work with.

Euler Angle Representation

In Euler angle representation, the axes of rotation are fixed to the object as opposed to being global axes. Consider x-y-z Euler angle representation: Rx(alpha) followed by Ry(beta), but around the local, rotated coordinate system. Using a prime symbol to represent rotation about a rotated frame, we have: Rx(alpha)R'y(beta). A y rotation around the rotated frame can be effected by Rx(-alpha)Ry(beta)Rx(alpha). So the result after the first two rotations is:

Rx(alpha)R'y(beta) = Rx(alpha) Rx(-alpha) Ry(beta) Rx(alpha) = Ry(beta) Rx(alpha)

The third rotation, Rz(gamma), is around the now twice rotated frame which can be effected again by undoing the previous stuff by Rx(-alpha)Ry(-beta) followed by Rz(gamma) and then redoing the previous rotations. Putting all three rotations together, and using a double prime to denote rotation about a twice rotated frame, gives us:

Rx (alpha) R'y (beta) R''z (gamma) = Ry(beta) Rx(alpha) Rx(-alpha) Ry(-beta) Rz(gamma) Ry(beta) Rx(alpha) = Rz(gamma) Ry(beta) Rx(alpha)

This system of Euler angles is really equivalent to the fixed angle system in reverse order. For example, z-y-x Euler angles are equivalent to x-y-z fixed angles. Therefore, the Euler Angle Representation has the same advantages and disadvantages that the Fixed Angle Representation has.

Euler Parameters

Euler showed that one orientation can be derived from another by a single rotation about an axis. Thus an orientation can be represented by a four-tuple of angle and (x,y,z) vector. The problem, however is that this representation cannot be used easily to interpolate between orietnations or to form a representation that represents concatenated rotations.

Quaternions

As shown, the above methods have problems when one tries to interpolate positions in those

representations. A better approach is to use quaternions to represent orientation. Quaternions were actually developed by Sir William Hamilton in 1843 as a successor to complex numbers. A quaternion is a four-tuple, [s,(x,y,z)] or [s,v], consisting of a scalar part ,s, and a vector part, v. Hence, the quaternion is an alternative representation to the Euler parameter representation. Importantly, it's in a form that can be interpolated and concatenated. Quaternion multiplication is defined as:

[s1,v1] * [s2,v2] = [(s1*s2-v1.v2),(s1*v2+ s2*v1+ v1Xv2)] where '.' denotes dot product and 'X' denotes cross product.

The inverse of a quaternion, [s,v], is obtained by negating its vector part and dividing both parts by the magnitude squared:

$q\text{-}1 = (1/||q||^2)*[s,-v]$

To rotate a vector, v, by a quaternion, q, treat the vector as [0,v] and:

$v' = Rot(v) = q^{-1}*v*q$

Notice how this guarantees that vector cross products are preserved:

Rot(v1) X Rot(v2) = Rot(v1 X v2)

Also notice that because all effects of magnitude are divided out, any scalar multiple of a quaternion gives the same rotation, similar to the homogeneous representation of points.

The interpretation of the quaternion representation is that of an axis of rotation and an angle. Euler in the 1600s (?) showed that any orientation can be represented by a single axis of rotation and a single angle. The unit quaternion representation of an orientation is in the form of

q = Rot((x,y,z),¿) = [cos(¿/2),sin(¿/2)(x,y,z)]

where ¿ is the angle and (x,y,z) is the axis of rotation. Notice that as far as the rotational interpretation is concerned, q = -q. In fact, q = k*q where k is an arbitrary non-zero constant.

The sphere of unit quaternions forms a subgroup, S3, of the quaternion group. We can rotate without speeding up by interpolating on the sphere, specifically along the great arc between the two quaternion points. A formula for spherical linear interpolation from q1 to q2 with parameter u varying from 0 to 1 can be obtained in two different ways. From the group structure we find:

Slerp(q1,q2,u) = q1(q1-1q2)u

while from 4D geometry comes:

Slerp(q1,q2,u) = (sin((1-u)*¿)/sin(¿))*q1 + (sin(u*¿)/sin(¿))*q2

where q1.q2 = cos(¿). The first is simpler for analysis while the second is more practical for applications. Notice that in the second one in the case u = 1/2, Slerp(q1,q2,u) = q1 + q2.

When interpolating between a series of orientations, slerping has the same problems that linear interpolation does between points in Euclidean space - first order discontinuities (see Appendix on

interpolation). In Schumaker's paper [ref], he suggests using cubic Bezier interpolation to smooth the interpolation between orientations. He uses a technique which automatically calculates reasonable interior control points to define cubic segments between each pair of orientations.

Assume for now that we need to interpolate between two-dimensional points in Euclidean space. In order to calculate the control point following any particular point, qn, take the vector defined by qn-1 to qn and add it to the point qn. Now take this point and find the average of it and qn+1. This point becomes an. Take the vector defined by an to qn and add it to qn to get bn. bn and an are the control points immediately before and after the point qn. A cubic Bezier curve segment is then defined by the point qn, an, bn+1, qn+1.

It should be easy to see how this procedure can be converted into the 4D spherical world of quaternions. Instead of adding vectors, rotations are concatenated. Averaging of orientations can easily be done by slerping to the half-way orientation.

Once the internal control points are computed, the De Castenue algorithm can be applied to interpolate points along the curve. In the Euclidean case of Bezier curve interpolation, the De Castenue construction procedure looks like the following.

The same procedure can be used to construct the Bezier curve in four-dimensional spherical space. To obtain an orientation corresponding to the u = 1/4 position alogn the curve, for example, the following orientations are computed:

```
p1 = slerp(qn,an,1/4)
p2 = slerp(an,bn+1,1/4)
p3 = slerp(bn+1,qn+1,1/4)
p12 = slerp(p1,p2,1/4)
p23 = slerp(p2,p3,1/4)
p = slerp(p12,p23,1/4)
```

where p is the orientation 1/4 along the cubic spline.

4.4 Camera Path Following

One of the simplest types of animation is that in which everything remains static except the camera. These are commonly referred to as walk throughs or flybys. Essentially the camera can be considered just as any other object as far as orientation and positioning is concerned. The user needs to construct a path through space for the observer to follow along with orientation information. The path is usually specified by key frame positioning followed by interpolation of the inbetween frames. There are various ways to deal with the view direction. For example, the center of interest can be held constant while observer position is interpolated along a curve (the view direction is the vector between the observer position and the center of interest). This is useful when the observer is flying over an environment inspecting a specific location such as a building. A path for the center of interest can be constructed, say from a series of buildings in an environment. Often, the animator will want the center of interest to stay focused on one building for a few frames before it goes to the next building. Alternatively, the center of interest can be controlled by other points along the observer path. For example, observer position for the next frame can be used to determine the view direction for the current frame. Sometimes this is too jerky. Some nth frame beyond the current frame can be used to produce a smoother view direction. The center of interest can also be attached to objects in the animation.

The specific method used to control view direction depends a lot on what the animation is trying to show. It can be handled in variety of ways.

Full orientation control is a bit more involved; controlling the observer position and view direction leaves one degree of freedom still to be determined: the observer tilt. One option is to just interpolate the view direction and then calculate the head up orientation and apply any tilt to that. The other is to specify another spline that controls the head-up position, but this can be hard to control.

4.5 Simple Key Frame System

The earliest computer animation systems were derivative of conventional hand-drawn animation systems in which positions of objects and the observer were specified at certain key frames. The positions and orientations were then interpolated to produce the frames between these keys. For now, we will only consider interpolation of positions; interpolating orientations involves issues particular to that application and are discussed as a separate topic in Section 1.3. Marcelli Wein and Hunger interpolated between 2-1/2D key frames.

Key frame animation is based on traditional animation techniques. Master animators would define and draw key frames that would essentially provide foundation pillars for the motion. Apprentice animators would then draw in the intermediate frames based on the key frames on either side of the intermediate sequence. This same idea was one of the earliest techniques implemented for two-dimensional computer animation. The ’master’ animator would draw key images and provide point-to-point correspondence information for interpolation programs which would then produce the intermediate images. The computer was an automated version of the apprentice animators. Although the computer could do the interpolation faster and more accurately than apprentices, it had to be provided with extensive information concerning what corresponded with what in the different key frames. These computer programs process two-dimensional information and therefore could not readily handle the three-dimensionality implied by animated figures the way the apprentices could.

4.6 A Simple Animation Language

Animation languages have the advantage of documenting motion, providing reproducible motions, providing algorithmic motion specification and being compact representations for motion. However, because they are essentially special purpose programming languages, in order to work with them, the animator needs to have a programmer’s mentality. Also, specifying complex motion with an animation language can be awkward.

Most are based on setting or changing object transformations over time. Light sources and the observer are usually animated the same way geometric objects are. Essentially an animation language provides the basic expression evaluation of any typical programming language along with some special constructions that have animation semantics.

```
principles
        basic - transformations over time
        programmability
                extension to languages, e.g., LISP, C
                        buy into variables, expressions, indirect referencing, subrouti
                readability, modifiability
                modular? graceful under complexity?
        interactivity
                script/batch based
                interactive setup of transformations
                interactive animation
```

```
        reactive
                conditional animation - reaction to circumstances that may arise during
                if (cond) then

miscellaneous
        instancing
        camera/view handling
```

Set/Change Commands

The fundamental facility that an animation language provides is a direct way to modify an object's transformation. Usually an object and its transformation representation are special data types that can be manipulated by these special commands. The most straightforward transformation manipulation commands are the ability to absolutely set or incrementally change something about the object's transformation. For example, simple commands may be of the form:

SET < object referent > TRANSLATION TO < x,y,z >
CHANGE < object referent > TRANSLATION BY < x,y,z >

The former is meant to be an absolute specification of the object's new translation. The latter is meant to be relative or incremental to the current value of the object's translation. The same type of thing can be done for the object's representation. The object's transformation refers to the object-to-world space transformation that was discussed earlier.

The Frame Counter

Besides intuitive access to the object's transformation, the animation language usually provides a special variable for time. Often this comes in the form of a frame number. Animation commands can then be scheduled to be active over some span of frames. For example, l SET < object referent > TRANSLATION TO < x,y,z > AT < frame number> CHANGE < object referent > TRANSLATION BY < x,y,z > FROM < fn1 > TO < fn2 > A looping construct is usually provided which loops over the frames of the animation. At the end of every loop, the objects which are active are rendered into an image.

Object Instances

An additional facility that is handled by some animation languages is the ability to perform such commands for some set of objects. For example,

```
CHANGE < object-set > TRANSLATION BY < x,y,z > FROM < fn1 > TO < fn2 >

or

FOR EACH OBJECT < object-set >
        CHANGE < object-set > TRANSLATION BY < x,y,z > FROM < fn1 > TO < fn2 >
        < fn1 > = < expression >
END
```

4.7 Hierarchical Structure Animation

Often it's the case that motion of one object is relative to the position of another object. The solar system is a good example. Assume that the moon rotates around the earth once every twenty-eight days and that the

earth rotates around the sun once ever three hundred and sixty five days. The resulting motion of the moon is rather complex (it's more complex if one considers that the sun is rotating around the center of the galaxy). The task is to set up a procedure that will be able to display the earth and moon in their correct positions given the day. Assume for simplicity that it's a two-dimensional problem.

Assume for now that the sun is at the origin and will stay that way, and that all of the objects are initially defined around the origin. Call Te the transformation matrix that translates the earth ninety three million miles out on the x-axis (and zero in y); call Re the transformation matrix that rotates the earth once every three hundred and sixty five days. The motion of the earth is simple:

E' = E*Te*Re (X)

To animate the earth, the matrix Re has to be updated with the current value of the day. As the value of the day increases, the rotation matrix Re will make the earth rotate about the sun.

Similarly, call Tm the transformation matrix that translates the earth two hundred twenty thousand miles out the x-axis and Rm the transformation matrix that rotates the moon once every twenty-eight days. The motion of the moon is a bit more complex:

M'=M*Tm*Rm*Te*Re (Y)

The first two transformation matrices, Tm and Rm, perform the rotation of the moon about the earth in the same way that Te and Re in Equation X performed the rotation of the earth about the sun. In Equation Y, Te and Re transform the moon, which at this point has been rotated about the earth, and rotate it about the sun along with the earth so that the motion of the earth and moon remain connected.

The important thing to notice is that the earth's transformations appear in the moon's transformations. If something were orbiting the moon, say a spaceship from the Earth, then its transformations would look like the following:

S' = S*Ts*Rs*Tm*Rm*Te*Re

Transformation hierarchies can be suitably represented by a tree structure. There are several ways to set up the structure. The one presented here is not the only way to do it. Consider the following tree structure: each node represents an object and each link represents a motion of one object relative to another. The root of the tree will be at the top, so a node in the tree moves relative to a node higher up in the hierarchy. See Figure Z.

Each node contains a pointer to an object file and a transformation matrix that transforms the object into a position such that the origin represents its point of attachment to the hierarchy. Each link of the tree structure represents the transformation that will animate that object. U sually this is a rotation, but it could be a translation just as well.

hierarchies in a simple animation language: groups attachments explicit parent-child relationships local coordinate systems v. global coordinate systems relative transformations

4.8 More Sophisticated Animation Languages

```
Utah '72 Catmull
        tree hierarchy
```

```
        arbitrary axes of rotations specified relative to parent node
        known/unknown lists
        known calls unknown with parameters

Hewitt ’73, MIT
        LISP based
        actors and message passing
        programming actors to respond to messages
        better complexity handling

DIRECTOR, ’76, Kahn, MIT
        LOGO turtle animation
        forward,backward, right, left, plan, make, receive, etc.

GRAMPS ’82, Berkeley, O’Donnell/Olsen
        VERB NAME AXIS FUNCTION
        only interactive animation, no script
        GROUP
        SCALE SPHERE SY D7
        assign values to dials
        value:steps
        frame - coordinate data sets for an object - metamorphosis


ASAS - Reynolds  MIT, III, SIG’82
        LISP based
        scripts
        actors
        animate block
        global and local operators
        Newtons - predefined sequence of values
        subworld and pov (point of view)
        pov same as 4x4 transform matrix

Mira ’83, CINEMIRA ’84, ANIMEDIT, Thalmans, Montreal
        laws - interpolation
        evolution - associates law with entity
        decor v. actor

FRAMES ’87 AT&T, Potmesil/Hoffert, SIG’87
        filters pipelined together (UNIX based)
        salt.frm|molecule|Euclid|shade|shade|camera|abuf
        sphere positioning, polygonalization, vertex shader, screen transformation, a b
        geometry, backface removal, shading, cameras, visible surfaces
```

4.9 Key Frame and Track-Based Animation Systems

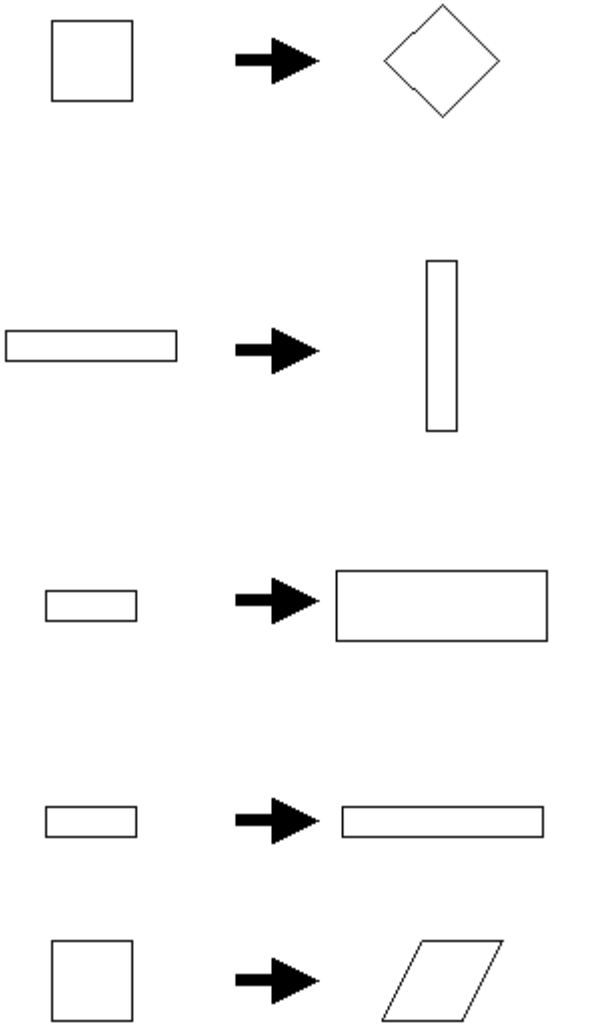
Track-based animation systems are a generalization of key frame systems. In track based systems each display parameter has its own ’track’ and its own keys. In key frame systems, all of the parameters would have to be specified in the key frames. In track-based systems, only those parameters which really have values that control interpolation are specified at any one point in time. This reduces the amount of information that has to be kept track of and focuses the interpolation only on the parameters that are important to the animator.

```
Moving-point constraint paper by Reeves

MUTAN ’83
        tracks - one per object
        generalizes from full key frame systems
```

```
TWIXT ’84, Gomex, OSU
        track per display parameter
        events - key frame for a particular track
        linear, parabolic, cubic interpolation with ease-in/ease-out
        blend objects
        hierarchies by explicit attachment one object to another
        absolute guiding
        object_name.field notation
        instancing
```

4.10 Shape Interpolation, Metamorphosis

Shape modification can be a flexible means for controlling animation, for example, the tail of a dog wagging, the swimming motion of a fish, or even human figure animation. The first question which needs to be answered is ’what is shape’? Or ’what transformations change the *shape* of an object’? Let’s consider linear transformations:



Which of the transformations shown above would you consider to have changed the *shape* of the figrue? I think we can agree that pure translation does not change the shape of an object. Does rotation? What about the difference between a diamond and a square? Are those two different shapes? We can probably agree that uniform scale does not change the shape of an object, but what about non-uniform scale? Most would

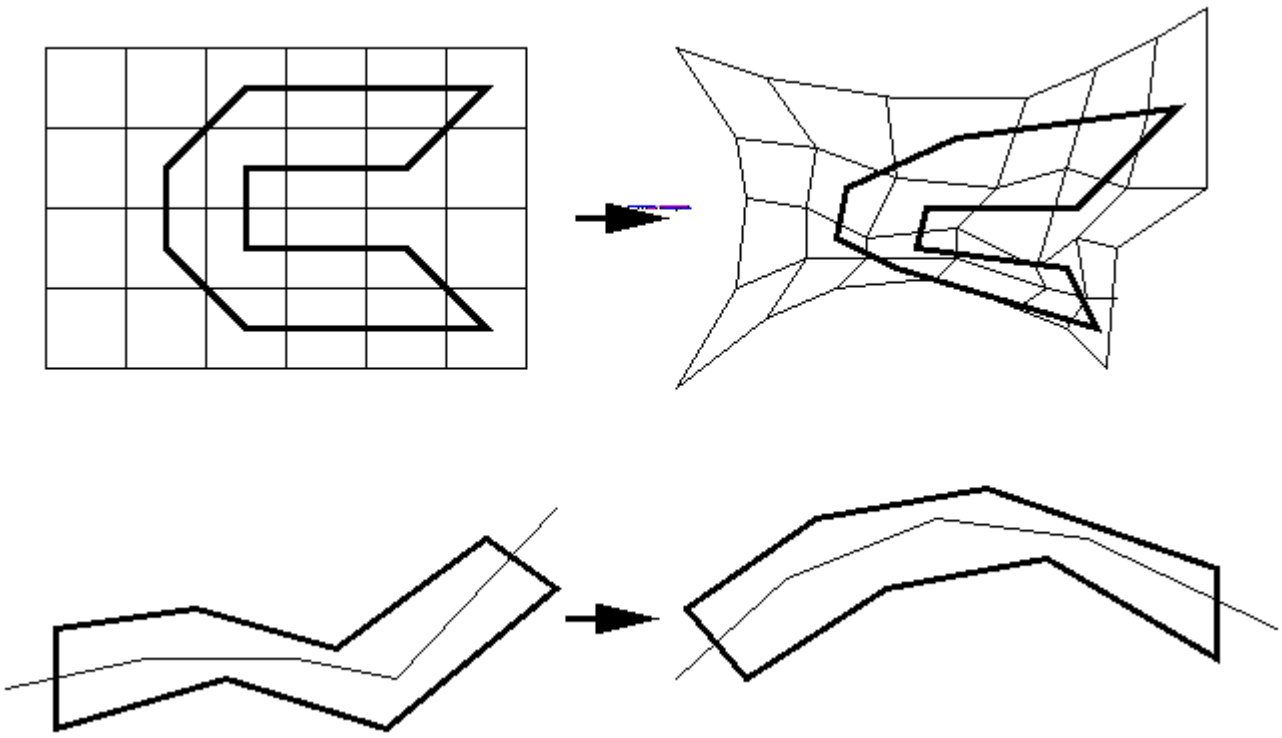
probably agree that non-uniform scale changes the shape of an object. Most would also agree that shearing changes the shape of an object.

The simplest class of transformations that (sometimes) changes the shape of an object is the affine transformations: transformations defined by a 3x3 matrix followed by a translation. Affine transformations can be used to model the squash and stretch of an object, the jiggling of a block of jello, and the shearing affect of an exaggerated stopping motion.

Two Dimensional Object Warping

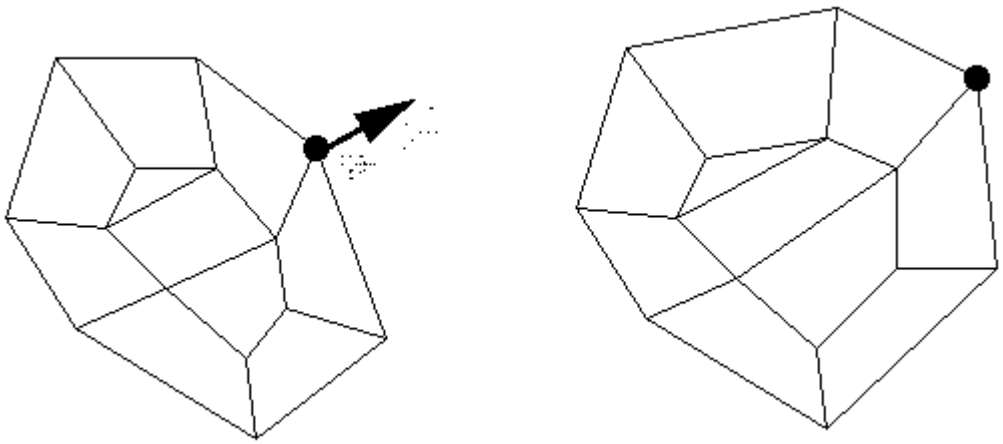
One of the first successful computer animations is in the film Hunger by Burtnyk and Wein and explained in their paper. In thefilm, a two-dimensional mapping was used to control animation. This essentially took a 2D coordinate system and allowed the user to reposition the coordinate grid points thus redefining 2D space. The object was then mapped back into this space using bilinear mapping.

Similar to Wein's technique, an object can be mapped onto a user-defined skeletal representation. The basic idea is the same - the skeleton is easier for the user to manipulate and then the object is remapped onto the modified skeleton. The user draws a wire (sequence of connected edges) through the object. The program calculates dividing lines at each vertex of the wire. Dividing lines at the extreme ends of the wire are perpendiculs to the respective end segments. Dividing lines at the interior vertices are bisectors of the angle made by the two adjacent segments. Each vertex of the object is falls within one semi-infinite cell; a cell is defined by a segment of the wire and its two associated dividing lines. A vertex position relative to the cell is established by drawing a line parallel to the cell's segment extending from one dividing line to the other. The distance between the segment and the parallel line through the vertex, together with the relative position of the vertex on that line are the local coordinates of that vertex with respect to that cell. After the skeleton has been modified by the user, each vertex of the object is relocated using newly computed dividing lines and the relative coordinates of the vertex. This has been implemented in 2D and has an obvious extension to 3D although branching in three-space is harder to consider.



The shape modification can also be performed on a vertex-basis instead of on a space-basis. A displacement for a seed vertex can be specified by the user and this displacement can be propagated to nearby vertices. The displacement can be attenuated as a function of the distance that the vertex to be displaced is away from the seed vertex. The *distance function* can be chosen to trade-off quality of results and computational complexity.

One such function would be the minimum number of edges connecting the vertex to be displaced from the seed vertex. Another such function would be the minimum distance traveled over the surface of the object to get from the vertex to be displaced to the seed vertex.

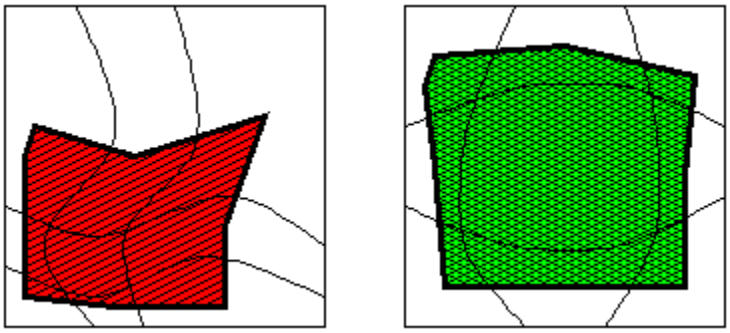


The attenuation function is a function of the distance metric. As an example, in Parent's paper, the user selected a function from a family of power functions to control the amount of attenuation. In addition, the user could select the maximum distance at which the displacement would have an affect.

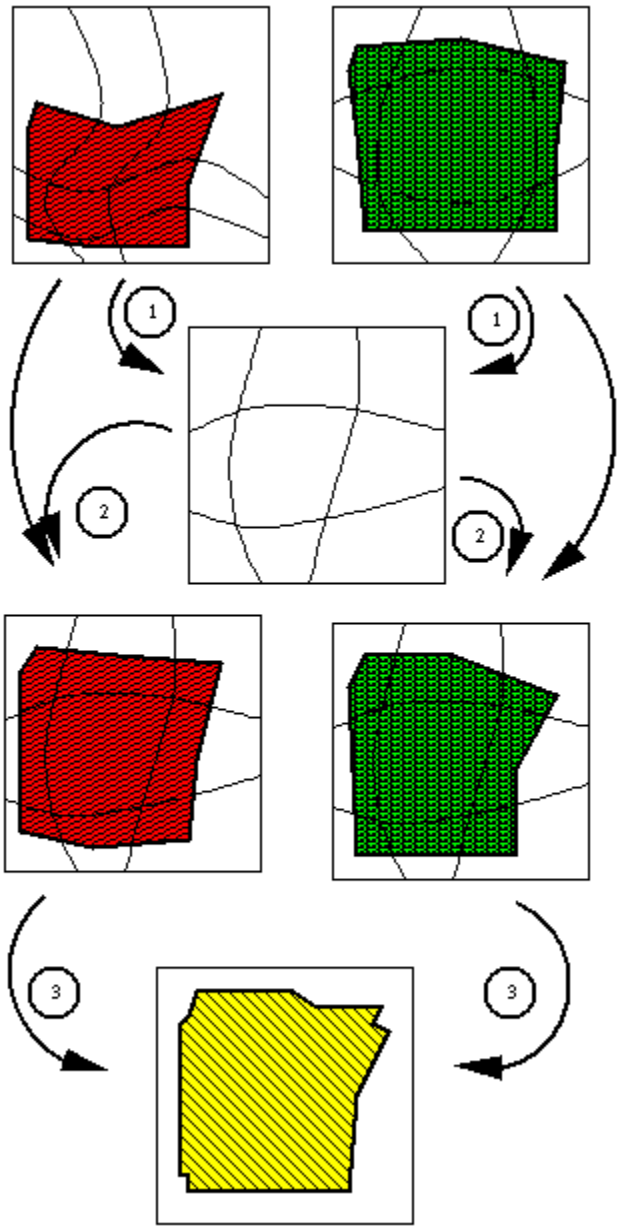
Two Dimensional Metamorphosis

The 2D case of image metamorphosis has become to be known as ’morphing’. Although really an image postprocessing technique and, as such, not central to the theme of this book, it has become so well-known and has generated so much interest that it demands to be treated on a book concerning computer animation.

In the static image case, morphing is accomplished by the user defining a curvilinear grid over each of the two images to be morphed. It is the user’s responsibility to define the grids so that elements in the images that he/she wants to correspond are in the corresponding cells of the grids. The user defines the grid by locating the same number of grid intersection points in both images in the same aspect ratio. A curved mesh is then generated using the grid intersection points as control points for an interpolation scheme such as Catmull-Rom splines.



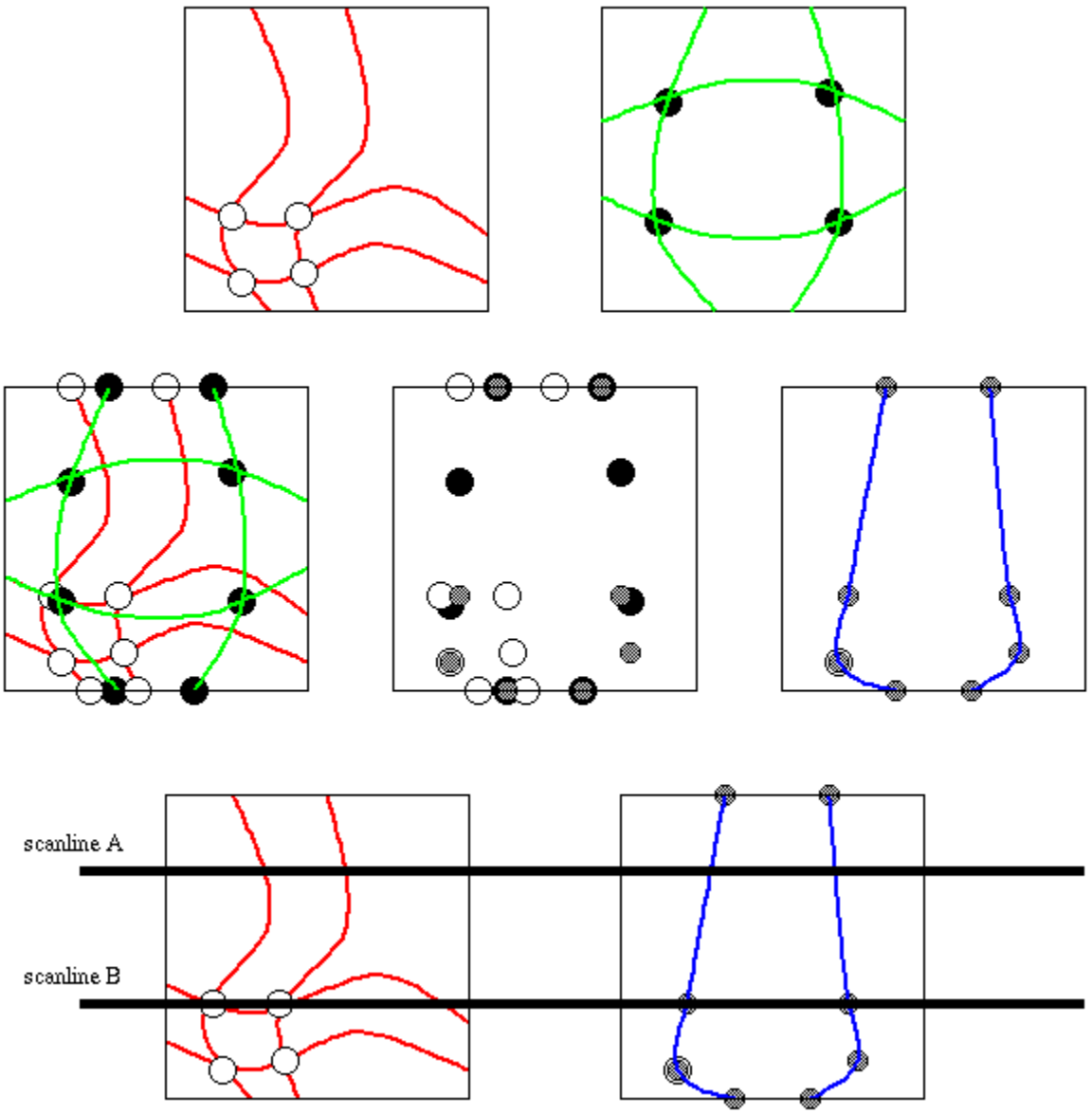
To generate an image between the two given images, say *t* along the way from the source image to the destination image, the grid vertices (points of intersection of the grid’s curves) are interpolated to form an intermediate grid. This interpolation can be done linearly, or grids from adjacent key-frames can be used to perform higher-order interpolation. The source and destination images are both warped according to the intermediate grid (as described below). A cross dissolve is then performed on a pixel-by-pixel basis between the two warped images to generate the final image.



The interpolation can be implemented in a two-pass procedure. Typically, the user is interested in transforming one image, which we'll call the source image, into the other image, which we'll call the destination image. In order to carry out the two-pass procedure, an intermediate grid is computed which, for each grid point, uses the x-coordingate of the source grid point and the y-coordinate of the destination grid point. We will also assume the curves are numbered left to right and will refer to a curve's number as it's index.

The first pass uses the source grid and intermediate grid to distort the source pixels in the x-direction. The columns of grid points are used to define cubic Catmull-Rom splines down both of the images. For each scanline, the x-intercepts of the curves with the scanlines are computed. For each pixel on the scanline in the source image, it's position relative to the x-intercepts is computed by computing the Catmull-Rom spline passing through the two-dimension space of (index, x-intercept) pairs. The integer values of x plus and minus one half, representing the pixel boundaries, can then be located in this space and the fractional index value recorded. In the intermediate image, the x-intercepts of the curves with the scanlines are also computed and for the corresponding scanline, the source image pixel boundaries can be mapped into the

intermediate image by using their fractional indices and locating their x-positions in the intermediate scanline. Once this is done, the color of the source image pixel can be used to color in destination image pixels using fractional coverage to effect anti-aliasing.



In the figure above, the top two images are the original two images to be morphed. The middle row of images show how the control points of the source grid (shown in white) are shoved horizontally to a position (shown in grey) directly underneath the control points of the destination grid (shown in black). The third image of the middle row shows the curves interplated through the interior intermediate control points. The third row shows two scanlines passing through the source image and the intermediate image. The first pass compresses and expands each scanline of the source image so that the grid intersection points line up with intersection points of the intermedicate image. Pixel colors are then resampled in the intermediate image.

In the second pass, this same procedure is now repeated in the y-direction in order to distort the intermediate image according to the destination grid. For each line of pixels down the image, y-intercepts with the grid curves are computed in the intermediate image and (index, y-intercept) pairs are recorded and

then located in the destination image using its grid.

In order to interpolate one image into another, the image grids are interpolated to form an inbetween grid. For a given frame in the interpolation, this becomes the destination grid. Colors are then computed for the destination image using the interpolated source image pixels. These colors are cross-dissolved with the target image colors.

Animated images are morphed by the user defining coordinate grids for various key images in each of two animation sequences. The coordinate grids are then interpolated over time so that at any one frame a coordinate grid can be interpolated for each sequence. Catmull-Rom interpolation can be used. The inbetween images are computed for each of the animated sequences to produce two images with associated grids. Once this is done, the procedure reduces to the static image case and proceeds accordingly.

Feature-Based Morphing: In a related paper, the correspondence between images is established by user-specified feature lines in both images. The user is responsible for establishing correspondence feature-line-pairs. Pixels in the source image are located relative to these feature lines by computing their perpendicular distance to the feature segments. Feature line positions are interpolated based on the interpolation parameter. Relative distance to feature lines is used to weigh the affect of that feature line's movement on that pixel.

Free-Form Deformations

Free-form deformations is essentially a 3D extension to Wein's technique. A 3D grid is used and object vertices are given coordinates relative to the grid. The grid can be manipulated by the user and the object vertices remapped into the distorted grid space. In Sederberg's original paper, Bezier interpolation was suggested as the interpolating function. B spline interpolating functions can also be used as well as Catmull-Rom interpolation. By animating the deformation over time, either by specifying key grids and interpolating intermediate grids, or by animating the grid vertices according to physical laws, the object deformation can be used to animate shape of the object. Successive papers gernalized on the rectilinear grid. In another paper, FFDs are strung together to form a structure through which an object can be passed and animated.

Three Dimensional Metamorphosis

Objects which share the same edge-connectivity can be interpolated on a vertex-to-vertex basis. 3D objects can also be metamorphosized by a variety of techniques. There are surface approaches which modify the vertex-edge connectivity (the 'topology') of the two object definitions until they match. Once that is done, a vertex-to-vertex blend of corresponding vertices can take place. The topologies can be matched by mapping each object to the surface of a sphere and then intersecting edges and using the combined intersected topology to replace both original topologies. A recursive approach can also be taken in which each object is reduced to two-dimensional sheets which are recursively subdivided. Edges and faces are added during subdivision to maintain topological equivalence. Surface-based approaches have trouble dealing with objects of differing geni. Volumetric approaches can also be used which avoid the problems of differing geni but require more computation because a surface definition has to be reconstructed at each in between stage. Minkowski sums, for example, can be used to define an intermediate volume for the objects from which a surface definition is formed.

Chen cut object into slices and uses center point of overlapping corresponding slices to equal up points on both slices. He linearly interpolates between pairs of points, one from each slice and then lofts a new

surface from the slices.

Kent 'blows up' objects to surface of sphere and intersects crossing edges (arcs) to form intersection vertices. He then add new points and points from other object to each object and remaps onto both objects. Now each object has same topology and he can do vertex-to-vertex interpolation.

Parent breaks each object into two sheets (e.g., front and back) and makes the same number of points on boundary of sheets. Then, recursively, he finds an existing path of edges between two vertices of one sheet and establishes a path of edges between the two corresponding vertices of corresponding sheet. He breaks each sheet into two sheets and recurse until each sheet is a single face. Now each object has same topology and once again, vertex-to-vertex interpolation can be performed.

Minkowski Sums translate one object all over surface of other object = Minkowski sum. He does linear interpolation of sizes of objects to do inbetween shapes of original object. THis may overcome differing topologies of starting shapes.

Distance Fields can be used for each object. For each voxel, calculate distance from surface of object to interpolate object, blend these distances to create new voxel space with new distances and then threshold to define surface of intermediate object.

4.11 Implicit Surfaces

Although not an animation technique, the use of implicit surfaces for object modeling can greatly enhance some animation effects which are not easily accomplished by other approaches. Implicit surfaces define an object as an isosurface of overlapping density functions. By convention, the density functions are cubic in distance from a defining structure. The most common implicit surface for animation is the metaball in which the defining structure is simply a center point. The density function is defined as 'one' at this center point and falls off monotonically until some limiting radius. Another structure is a lines segment which gives rise to a cylinder with spherical ends. Varying the radius over the length of the straight line produces a spherically capped cone. The user overlaps these various elements and then either a Marching Cubes type algorithm or a ray tracer can evaluate the isosurface given some threshold. To produce animation, the structures are animated by any of the techniques covered in this book and then the isosurface is reevaluated. Animations produced by using implicit surfaces are very organic looking and are well-suited to cartoon-like character animation.

Go back to Table of Contents
Go to Previous Chapter: Chapter 3. Display Considerations
Go to Next Chapter: Chaptger 5. Algorithms and Models to Fully Specify Motion

---

Rick Parent -- *(parent@cis.ohio-state.edu)*
last updated 8/1/98